A Model for Assessing the Liability of Seemingly Correct Software

Jeffrey M. Voas* Larry K. Voas† Keith W. Miller‡

Abstract

Current research on software reliability does not lend itself to quantitatively assessing the risk posed by a piece of life-critical software. Black-box software reliability models are too general and make too many assumptions to be applied confidently to assessing the risk of life-critical software. We present a model for assessing the risk caused by a piece of software; this model combines software testing results and Hamlet's probable correctness model [2]. We show how this model can assess software risk for those who insure against a loss that can occur if life-critical software fails.

Keywords

Software reliability, probable correctness, probability of failure, software safety, fault, insurance, risk assessment, failure, peril, hazard.

1 Introduction

Software liability issues can be characterized by two questions: (1) who is liable when a piece of software fails catastrophically and (2) how great is their liability? The questions of liability are legal and ethical quandaries, but they also introduce technical questions. This paper is concerned with quantifying the financial risk associated with a piece of software, a technical question closely associated with the second question above.

If a piece of software never fails, then questions of liability do not arise; but determining **a priori** that software will never fail is, in general, impossible. In order to quantify the financial risk of software, we need numbers that describe the probability that the software will fail catastrophically and the cost of the resulting catastrophe. In this paper we will define a software failure as when software does not perform as specified and this aberrant behavior threatens life. We will use the probable correctness model introduced by Hamlet [2] to quantify the probability of failure, and we will show how this figure can be combined with numbers estimating the cost of such failures. We will not describe how insurance underwriters estimate the cost of a catastrophe.

2 How Insurance Companies Assess Risk

Risk can be divided into two types: *speculative* and *pure*. Speculative risk can either result in a loss or a profit. An example is buying company stock. With pure risk the only potential consequence is loss.

*Supported by a National Research Council NASA-Langley Resident Research Associateship. Author's correspondence address is: Reliable Software Technologies Corporation, Penthouse Suite, 1001 North Highland Street, Arlington, VA 22201, (703) 276-1219, FAX (703) 527-0200.

†Northwestern Mutual Life Insurance Company.

‡Supported by NASA-Langley Research Center Grant NAG-1-884. Author's correspondence address is: Dept. of Computer Science, College of William & Mary, Williamsburg, VA 23185, (804) 221-3464.

An example is automobile collision insurance.

Insurance gives companies that produce goods and services an opportunity to order their business affairs so that a certain cost (*premiums*) are substituted for an uncertain cost (potentially disastrous loss of unacceptable magnitude). The ability to acquire insurance permits security and thus continued economic activity in an environment containing risk. Failure to substitute risk via the payment of premiums means retention of the potential adverse consequences. Few prudent businesses are willing to assume such a risk.

But what do insurance companies (risk assumers) need in order to offer the security of insurance? First, they seek to quantify the risk. Risk quantification evaluates uncertainties and places a dollar amount on the risk to be assumed. *Perils* are the cause of pure risk. Examples of perils include floods, fire, and disease. Underwriting inquirers also look beyond the peril to its cause (*hazards*) which precipitate the resulting loss. Examples of hazards include bad weather patterns for the peril of flood, poor electrical wiring for the peril of fire, and poor health habits for the peril of disease. Thus the providers of insurance need tools to measure the risk/hazards so they can mathematically calculate premiums. Without such tools, the insurance provider cannot prudently assume risk and accurately set rates.

The early writers of marine insurance estimated the risks of ship and cargo loss by taking into account weather patterns for specific routes at specific times, the patterns of known pirate activity, and the success and failures of the crew. Underwriter accuracy in assigning a premium/risk ratio is enhanced if a large number base of previous shipping experiences is available.

Suppose that an underwriter assigns risk numbers from 1 to 10 (smallest risk to greatest risk) on each of the assumed equal categories of the perils: weather, pirates, previous crew/ship experiences. For example, assume that a winter voyage calls for a weather rating of 9, pirate activity calls for a 2, and the crew/ship have an excellent rating of 1. In this case, 12 is the combined risk out of a possible total of 30. A premium of 12/30 of the insured value would be assessed as well as some value for underwriter profit.

The relatively high premium for this proposed trip could be justified if the shipper anticipates inflated profits due to a winter delivery. However the shipper might decide to sail later when the weather peril decreases. A final option is for the shipper to sail without protection and retain all risks. Many decisions were involved in these crude tools of early risk management. But the concept of seeking data to turn total uncertainty into reasonable certainty is still the goal of those who seek and sell insurance.

In summary, any technique or tool that enables an insurer to more accurately assess a risk makes cost effective insurance more available. The next section describes such a tool for businesses who want to insure against catastrophic software failure.

3 Quantifying Software Risk

Insurance companies can “play the percentages” best when they have extensive prior experience that is directly relevant to a certain risk. Prior knowledge allows them to predict with reasonably high confidence the likelihood of a loss.

Software risk is a pure risk: the software is expected to work correctly; the only “surprise” is the unwelcome surprise of a software failure. Software is a relatively new product, particularly software that could be associated with a loss-of-life. There has not been a lengthy historical time frame over which software systems can be evaluated to accumulate prior knowledge. And each software system is unique; generalized risk analysis from previous systems will not necessarily apply to a current project. This is a drawback of black-box software reliability models based on error histories: they often overgeneralize based on the results from previous experiences.

Also, critical software systems are generally one-of-a-kind; there may be many copies, but they are all identical. Statistics such as percentage of defective parts out of a large lot of parts do not apply here. N-version programming was an idea explored in the 1980's to show that multiple implementations of a specification produced a more reliable overall system. The results from experimentation using multiple implementations have been disappointing; N-version programming is based on the assumption that software failures occur in different versions independently, and this assumptions has not been substantiated [3].

To assess software risk, we need to quantify the reliability of the software. There are many software reliability models [7]; many of these are *error history models*, which estimate future software failures by tracking the history of errors discovered previously. Different error history models often assign different reliability estimates to the same set of data and it is inconclusive as to which model is the most accurate for a given piece of software. Also, error history models do not work well for very high reliability estimates. In short, error history reliability models are of limited use when insurers wish to assess the risk of critical/ultra-reliable software. In the rest of this section we introduce a reliability formula based on black-box testing and probability theory; we contend this this formula embodies a theoretically defensible and utilitarian method for producing consistent estimates of software reliability.

3.1 Probable Correctness and Software Reliability

§3.1 discusses a technique that estimates a maximum failure rate for a piece of software; using this technique, we establish a value $\hat{\gamma}$, which acts as a upper bound on the likely probability of failure for this software. This technique requires that the user know what the distribution of inputs will be for the software. The user executes random black-box tests drawn from this distribution, and determines for each test whether or not the software's output constitutes a failure. This determination is the most expensive and error-prone part of most testing efforts. If an automated test oracle is available to give a correct answer, extensive testing becomes possible; however, such oracles are rare. (Some examples do exist; for example, if a new program is a re-implementation that is supposed to exactly mimic an old program, the old program can function as an automated oracle.)

After T tests are executed without revealing any faults, the user can set a confidence level C, and the formula given in equation 1 on page 5 can be used to calculate $\hat{\gamma}$ such that with confidence C we can state that the actual probability of failure of a single execution is $\leq \hat{\gamma}$ [2].

Randomly generated black-box testing is an established method of estimating software reliability [8, 13]. Unfortunately, as software applications have required higher and higher reliabilities, practical limitations of black-box testing have become increasingly problematic. These practical problems are particularly acute in life-critical applications, where requirements of 10^{-7} failures per hour of system reliability translate into a required probability of failure (**pof**) of perhaps 10^{-9} or less for each individual execution of the software [6]. In the rest of this paper we will refer to software with reliability requirements of this magnitude as *ultra-reliable* software.

The **pof** of a program is conditioned on an input distribution. An input distribution is a probability density function that describes for each legal input the probability that the input will occur during the use of the software. Given an input distribution, the **pof** is the probability that a random input drawn from that distribution will cause the program to output an incorrect response to that input.

Even if ultra-reliable software can be in theory achieved, we cannot comfortably depend this achievement unless we can establish that reliability in a convincing, systematic, and scientific manner. As pointed out in [4], black-box testing is impractical for establishing these very high reliabilities. In general, by executing T random black-box tests, we can estimate a probability of failure in the neighborhood of $1/T$ when none of the tests reveals a failure [5]. If the required reliability is in the ultra-reliable range, random black-box testing would require decades of testing before it could establish a reasonable confidence in this reliability, even with the most sophisticated hardware. Based on these impracticalities, some researchers contend that very high reliabilities cannot be quantified using statistical techniques [1].

Hamlet has derived an equation to determine what he calls "probable correctness" [2]. When T tests have been executed and no failures have occurred, then:

$$C = \text{Prob}(\theta \leq \hat{\gamma}) = 1 - (1 - \hat{\gamma})^T \tag{1}$$

where θ is the true **pof**, $0 < \hat{\gamma} \leq 1$, and C is the confidence that $\theta \leq \hat{\gamma}$. [1] We can rearrange this formula to:

$$\hat{\gamma} = 1 - (1 - C)^{(1/T)} \tag{2}$$

With this formula, we have an upper bound on an estimate of the **pof** in $\hat{\gamma}$. This provides us with the maximum risk that we take when we execute the software a single time.

Persons interested in the results of equation 2 include software developers, software testers, and persons insuring against the failure of the software system. The developer gains a feeling for how confident he is currently given the degree of testing of the software project. The tester gains confidence in the software which the testing has produced to date, and the tester can use equation 2 to find out much additional successful testing will be needed until the **pof** of the software is less than some preset threshold with some C. An insurer gains an ability to assess the risk that a piece of software provides. For the insurer, §3.2 shows how to apply equation 2 and determine an insurance premium against software failure.

Equation 2 presents an interesting dilemma. While C is fixed and T increases, $\hat{\gamma}$ decreases; this translates into a decreased risk. But as the risk decreases, the likelihood of ever revealing the existence of a fault without enormous amounts of additional testing also decreases. What this means is that as testing continues without observing failures, we learn that *if* faults are hiding in the code, we are unlikely to catch them during testing. Faults that hide during testing are a nightmare for those who develop life-critical code.

3.2 Probable Correctness, Software Liability, and Insurance

In order to assign a premium, insurance companies use a simple formula where τ represents the maximum financial loss caused by some catastrophic event being insured against and χ represents the probability of that event occurring. Then the premium p assessed is simply

$$p = (\tau \cdot \chi) + \text{Profit}. \tag{3}$$

When we apply this to software and use the $\hat{\gamma}$ derived from equation 1 and T successful tests, this becomes:

$$p = (\tau \cdot [1 - (1 - C)^{(1/T)}]) + \text{Profit}. \tag{4}$$

Table 1 shows how various values for C and T affect p when τ is set at $1,000,000. Note that the premium (without Profit) shown in Table 1 is the premium that should be assessed for a single execution of the software that has been based on the previous amount of successful testing T and the preset confidence C. Thus we must scale p according to the expected amount of use of the software during the time period of the insurance. For instance, if during the period of the insurance, the software will be executed n times, we will adjust the cost of p on the k^{th} $(1 \leq k \leq n)$ execution to be:

$$p_k = (\tau \cdot [1 - (1 - C)^{(1/(T+(k-1)))}]) + \text{Profit}. \tag{5}$$

[1] Hamlet calls C a measure of probable correctness, but it would be called a confidence if the equations were cast in a traditional hypothesis test.

$p-$ Profit	τ	C	T	$1-(1-C)^{(1/T)} = \hat{\gamma}$
\$500,000	\$1,000,000	0.999	10	0.499
\$67,114	\$1,000,000	0.999	100	0.067
\$6,891	\$1,000,000	0.999	1000	0.00689
\$7	\$1,000,000	0.999	1,000,000	0.0000069
\$0.70	\$1,000,000	0.999	10,000,000	0.00000069
\$0.07	\$1,000,000	0.999	100,000,000	0.000000069
\$369,003	\$1,000,000	0.99	10	0.369
\$45,004	\$1,000,000	0.99	100	0.045
\$4,590	\$1,000,000	0.99	1000	0.00459
\$5	\$1,000,000	0.99	1,000,000	0.00000460
\$258,064	\$1,000,000	0.95	10	0.258
\$29,498	\$1,000,000	0.95	100	0.0295
\$2,994	\$1,000,000	0.95	1000	0.00299
\$3	\$1,000,000	0.95	1,000,000	0.00000299

Table 1: Sample values obtained using Equation 4.

For all n executions, p will be:

$$p = \sum_{i=1}^{n} [(\tau \cdot [1-(1-C)^{(1/(T+(i-1)))}]) + \text{Profit.}] \tag{6}$$

We assume that if the software ever fails during the insured period, execution of the software will immediately cease. This means that the maximum loss for the insurer is τ.

As can be seen in Table 1, virtually no one could ever afford the cost of insuring software for a single execution that has only been tested 10 times. Furthermore, the cost of insuring software for multiple executions that has received such minimal testing quickly surpasses τ in equation 6, making such a premium assessment useless. To reduce the premium, additional testing must be performed. The trade-off between testing costs and premium can be optimized.

As shown in Table 1, when we move from 99% confidence to 95% confidence, the insurer accepts more of the risk himself, and by demanding less confidence from the successful testing, is able to charge a lower premium. If the insurer wishes to charge a lower premium with a lower confidence, then the insurer must observe more successful tests. This is the trade-off between confidence and testing: when confidence is diminished, the number of successful tests must increase, or else the premium will increase. When confidence and number of successful tests increase, then the premium can decrease.

4 Concluding Remarks and Future Research

This paper has shown how the probable correctness model can be used to estimate the maximal **pof** of a piece of software given a confidence C for this estimate and T successful executions. By having an estimate of the maximal **pof**, we have an assessment of the maximum risk that we estimate the software poses. With an upper bound estimate of the risk, we can begin to explore how to insure against the potentially disastrous consequences of software failure. The upper bound allows us to calculate an insurance premium against software failure, given that the maximum insurance protection is preset.

We think that issuers of liability insurance can better assess the premiums they charge for software liability insurance by using the fact that a piece of software is known to have not failed. By combining this fact with a confidence, an upper bound on the risk is attainable via a simple equation. This is beneficial for both buyers and sellers of software liability insurance.

As shown in Table 1, the costs of insurance are prohibitive for software that is frequently executed, say one thousand times per day. Even in the scenario where we tested one million times successfully with 95% confidence, the premium would be at least at least $3,000 per day before profit is calculated. This cost reflects the difficulty of assessing ultra-reliability using statistical methods [1]. The weakness of black-box testing is that it is ineffective against failures that occur very infrequently.

Although black-box testing is thwarted by infrequent failures, such testing in concert with other analysis may still allow more precise reliability assessment. Testability analysis [12, 9, 10, 11] is a new technique that attempts to quantify the likelihood that faults can hide from testing. By combining software testability information with successful software testing information, we expect to be able to substantially decrease the premiums that we have shown in this paper for software systems that show a tendency to reveal faults during testability analysis. Research into the application of testability analysis to liability analysis is ongoing.

References

[1] R. Butler and G. Finelli. The infeasibility of experimental quantification of life-critical software reliability. In *Proceedings of SIGSOFT '91: Software for Critical Systems*, pages 66–76, New Orleans, LA., December 1991.

[2] Richard G. Hamlet. Probable Correctness Theory. *Information Processing Letters*, pages 17–25, April 1987.

[3] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumptions of independence in multiversion programming, *IEEE Trans. on Software Engineering*, SE-12:96–109, Jan. 1986.

[4] D. R. Miller. Making Statistical Inferences About Software Reliability. Technical report, NASA Contractor Report 4197, December 1988.

[5] K. Miller, L. Morell, R. Noonan, S. Park, D. Nicol, B. Murrill, and J. Voas. Estimating the Probability of Failure When Testing Reveals No Failures. *IEEE Trans. on Software Engineering*, 18(1):33–44, January 1992.

[6] I. Peterson. Software failure: counting up the risks. *Science News*, 140(24):140–141, December 1991.

[7] C. V. Ramamoorthy and F. B. Bastani. Software Reliability - Status and Perspectives. *IEEE Transactions on Software Engineering*, SE-8:543–371, July 1982.

[8] T. A. Thayer, M. Lipow, and E. C. Nelson. *Software Reliability*(TRW Series on Software Technology, Vol. 2). New York: North Holland, 1978.

[9] J. Voas, L. Morell, and K. Miller. Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2), March 1991.

[10] J. Voas and K. Miller. The Revealing Power of a Test Case. *J. of Software Testing, Verification, and Reliability*, To appear in 1992.

[11] J. Voas. *A Dynamic Failure Model for Performing Propagation and Infection Analysis on Computer Programs*. PhD thesis, College of William and Mary in Virginia, March 1990.

[12] J. Voas. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, To appear August, 1992.

[13] S. N. Weiss and E. J. Weyuker. An extended domain-based model of software reliability. *IEEE Trans. on Software Engineering*, 14(10):1512–1524, October 1988.